Game of Life

Introduction:

I wrote a game of life application that takes an arbitrary size board and outputs the next phase in Game of life according to the game of life rules. It also provides the option to output more than one generation (so not just the next phase).

How to run:

java -jar dist/Game\_of\_Life.jar 8 6 1 input.txt

Game\_of\_Life.jar is the program, 8 is the length of the 8x6 grid, 6 is the height of the 8x6 grid. 1 is the number of generations to calculate. Each generation is output. The last argument is the input file that Game\_of\_Life is finding the next phase for.

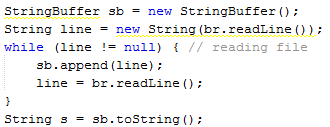
The input file must be a set of lines of 1s and 0s. The 1s represent alive cells in the grid, the 0s represent dead cells in the grid. Each line in the input file is the same length.

Design:

First I initialize the variables. The four argument needed to run the program are listed above. xSize takes the length of the board, ysize takes the height of the board, gen takes the number of generations and br is the inputStream for the file specified in the fourth argument.



Then I created a StringBuffer to get all the characters in the input file.

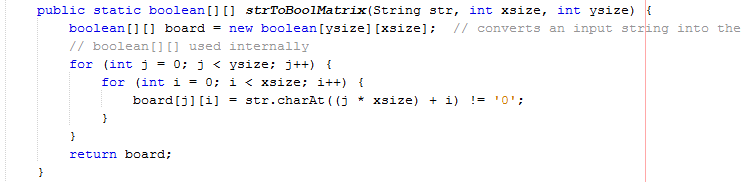


The input is a grid of cells. Each cell has one of two states: dead or alive. Therefore, I choose to represent the board as an array of array of booleans. I delegated the task of creating the board inside a method strToBoolMatrix:



strToBoolMatrix takes the string representation of the grid, the grid length, “xsize” and the grid height, “ysize”.

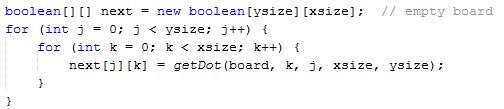
strToBoolMatrix iterates through each cell in the grid and checks whether it’s a 1 or 0:



After creating the board GOL iterates through each generation.



Each generation goes through each cell in the current board and creates a new board based on the game of life rules.



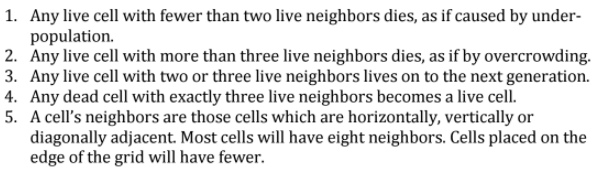
When the new grid is complete the current board is set to the new board



and the new board is output using boardToString.



getDot and getNeighborCount are the heart of the program. getDot returns the value of the cell, alive or dead. getNeighborCount is used by getDot, it calculates how many neighbors there are. There are 5 rules to the game of life:

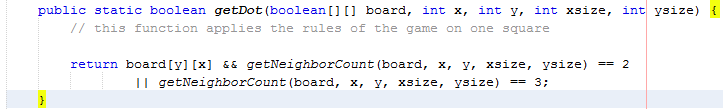


The first 4 rules can be boiled down into two rules:

1. A live cell must have 2 or 3 neigbors to stay alive; otherwise, it dies
2. A dead cell must have 3 neighbors to become alive; otherwise, it stays dead.

These two rules can be boiled down to:

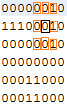
1. Any cell with 3 neighbors is alive
2. Any live cell with two neighbors is alive



getNeighborCount calculates rule 5 in the game of life:

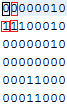


There are 8 possible neighbors for a cell:



This cell has 3 alive neighbors so it’s going to become alive.

There are also cells with less neighbors:



The logic in getNeighborCount determines what neighbors exist and are alive based on the position of the cell and the board representation.

The cell above has two neighbors, but it is dead so it stays dead.

getNeighborCount can be thought of in three parts:

1. neighbors above the current cell
2. neighbors adjacent to the cell
3. neighbors below the cell

To calculate how many live neighbors exist above the cell we first must determine if there are any neighbors above the cell:



y is the row of the current cell. If y=0 then the row is on top; hence, no neighbors exist above the cell; hence, no more need to calculate how many neighbors are alive. If y>0 than there are cells above it so we find the value of the

left:

 we subtract 1 from x since indexes increase as you go to the right

middle:



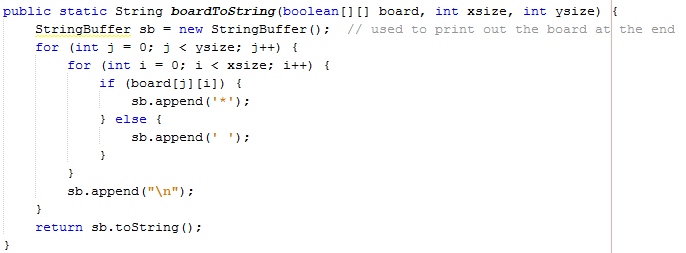
right:

]

cells to count the neighbors.

part 2 and 3 use similar logic.

boardToString takes the board representation and outputs it in a user-friendly way where live cells are denoted with astericks and dead cells are denoted by a space.



Output:



Testing:

To test this project open the project in netbeans, right click the test file and select Test file.

